**ЛАБОРАТОРНАЯ РАБОТА №1**

**«ИССЛЕДОВАНИЕ ЛИНЕЙНЫХ И НЕЛИНЕЙНЫХ СТРУКТУР ДАННЫХ. ЛИНЕЙНЫЕ СПИСКИ И БИНАРНЫЕ ДЕРЕВЬЯ ПОИСКА»**

* 1. **Цель работы**

Исследовать возможности применения линейных и нелинейных структур данных – линейных списков и бинарных деревьев поиска – для хранения, поиска и обработки информации. Приобрести практические навыки создания и использования классов, реализующих списки и бинарные деревья поиска и исследовать эффективность этих структур данных при выполнении операций добавления удаления и поиска данных.

* 1. **Вариант задания**

Вариант 8

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ключевое поле | N1 | N2 | N3 | N4 | N5 |
| Абонент | 30 | 500 | 1900 | 4000 | 7500 |

* 1. **Ход выполнения работы**

Была написана программа на языке C++, которая реализует классы для работы со списком и с бинарным деревом поиска. Текст программы представлен в приложении А.

В качестве информационного поля для дерева и списка используется структура Data, которая содержит 5 полей, а также перегруженные операторы ввода и вывода. Для заполнения информационных полей были написаны функции для генерации случайных значений. Для описания элемента списка используется класс ListNode, он содержит информационное поле и указатель на следующий элемент списка. Для описания узла дерева используется класс TreeNode, он также содержит информационное поле и указатель на левого и правого потомка.

Для работы с самими структурами данных написаны соответствующие классы List и Tree. Для них определены методы добавления, поиска и удаления. Также для них определен метод, который генерирует файл для пакета утилит визуализации графов. В этом файле описываются все связи необходимые для визуального построения линейного списка и бинарного дерева поиска.

В таблице 1.1 отражено время выполнения операций при разном количестве элементов в списке и дереве.

Таблица 1.1. Результаты проделанной работы

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Время N | № | Список | | | Дерево | | |
| Добавление | Поиск | Удаление | Добавление | Поиск | Удаление |
| N1 = 30 | 1 | 0.0045 | 0.0601 | 0.0347 | 0.005 | 0.006 | 0.004 |
| 2 | 0.0105 | 0.0507 | 0.051 | 0.013 | 0.003 | 0.003 |
| 3 | 0.0128 | 0.059 | 0.0388 | 0.009 | 0.005 | 0.004 |
| 4 | 0.0075 | 0.0659 | 0.0339 | 0.012 | 0.003 | 0.003 |
| 5 | 0.006 | 0.0309 | 0.0503 | 0.017 | 0.002 | 0.002 |
| Ср. | 0.008 | 0.053 | 0.042 | 0.011 | 0.004 | 0.003 |
| N2 = 500 | 1 | 0.0058 | 0.0037 | 0.0049 | 0.005 | 0.002 | 0.002 |
| 2 | 0.0043 | 0.0538 | 0.0174 | 0.013 | 0.002 | 0.001 |
| 3 | 0.0161 | 0.0493 | 0.0455 | 0.009 | 0.001 | 0.002 |
| 4 | 0.0073 | 0.031 | 0.0643 | 0.015 | 0.002 | 0.002 |
| 5 | 0.0055 | 0.0161 | 0.0128 | 0.008 | 0.002 | 0.002 |
| Ср. | 0.009 | 0.054 | 0.046 | 0.011 | 0.002 | 0.002 |
| N3 = 1900 | 1 | 0.0417 | 4.6251 | 1.807 | 0.154 | 0.084 | 0.076 |
| 2 | 0.0598 | 1.4631 | 1.4631 | 0.133 | 0.130 | 0.086 |
| 3 | 0.0455 | 2.0162 | 0.7815 | 0.103 | 0.075 | 0.066 |
| 4 | 0.0471 | 1.0861 | 4.6217 | 0.137 | 0.080 | 0.129 |
| 5 | 0.0491 | 0.1508 | 1.5474 | 0.086 | 0.073 | 0.055 |
| Ср. | 0.049 | 1.868 | 2.049 | 0.123 | 0.088 | 0.083 |
| N4 = 4000 | 1 | 0.0417 | 4.7803 | 1.8719 | 0.140 | 0.086 | 0.103 |
| 2 | 0.0747 | 2.5566 | 5.4383 | 0.115 | 0.055 | 0.080 |
| 3 | 0.0696 | 1.9187 | 2.8572 | 0.109 | 0.083 | 0.067 |
| 4 | 0.0588 | 0.5316 | 6.4647 | 0.112 | 0.060 | 0.093 |
| 5 | 0.0891 | 0.979 | 1.8547 | 0.133 | 0.084 | 0.071 |
| Ср. | 0.067 | 2.153 | 3.697 | 0.122 | 0.074 | 0.083 |
| N5 = 7500 | 1 | 0.0415 | 3.4791 | 1.7372 | 0.152 | 0.106 | 0.058 |
| 2 | 0.0529 | 2.0095 | 3.7955 | 0.167 | 0.059 | 0.116 |
| 3 | 0.0535 | 0.333 | 4.1635 | 0.128 | 0.073 | 0.092 |
| 4 | 0.0586 | 0.0276 | 0.4434 | 0.138 | 0.089 | 0.090 |
| 5 | 0.0502 | 0.9128 | 2.0694 | 0.161 | 0.098 | 0.074 |
| Ср. | 0.051 | 1.352 | 2.442 | 0.149 | 0.085 | 0.086 |

Было проведено добавление 100000 элементов в линейный список и бинарное дерево поиска. Результаты изображены на рисунке 1.

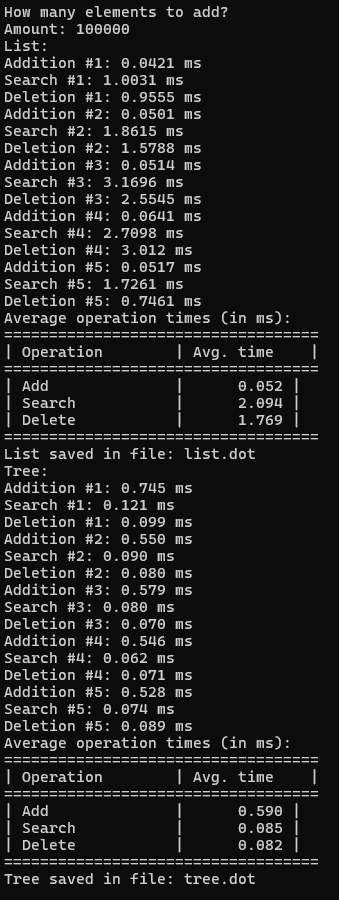


Рисунок 1 – Результаты операций добавления, удаления и поиска для линейного списка и дерева при 100000 элементов

При помощи пакета утилит для визуализации графов была выведена структура бинарного дерева поиска на 50, 150 и 500 элементов. Это изображено на рисунках 2 – 4.

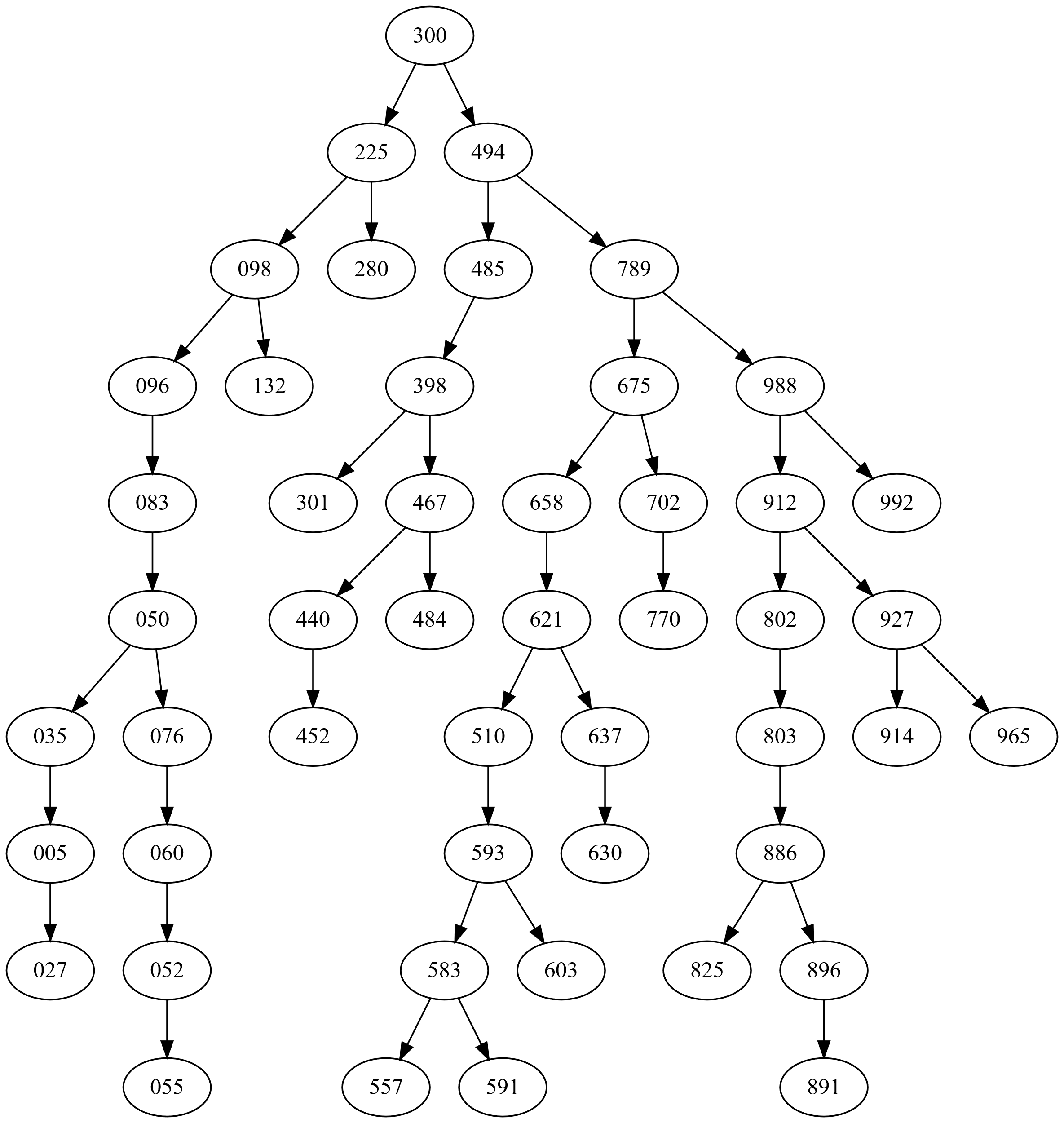


Рисунок 2 – Дерево из 50 элементов

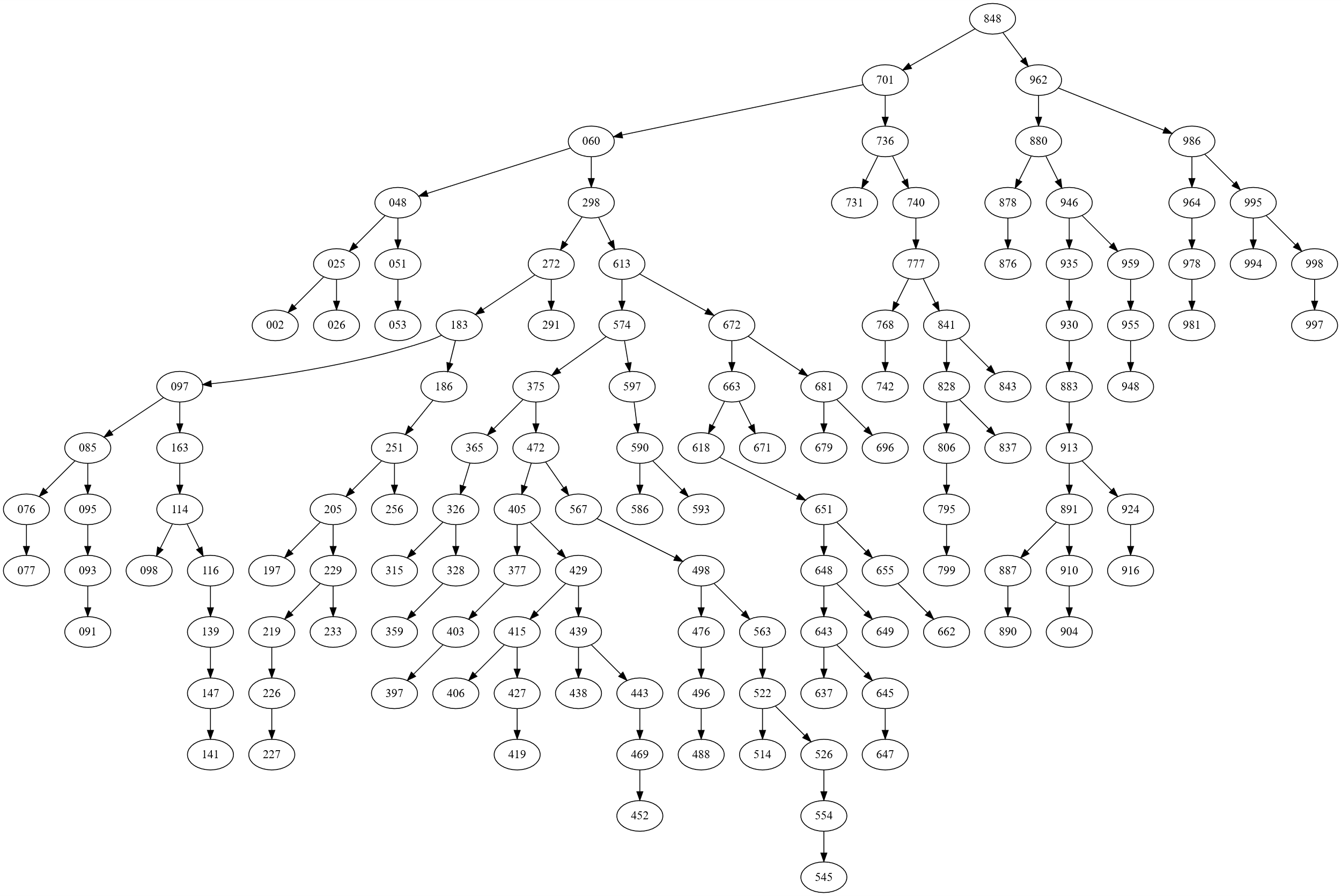


Рисунок 3 – Дерево из 150 элементов

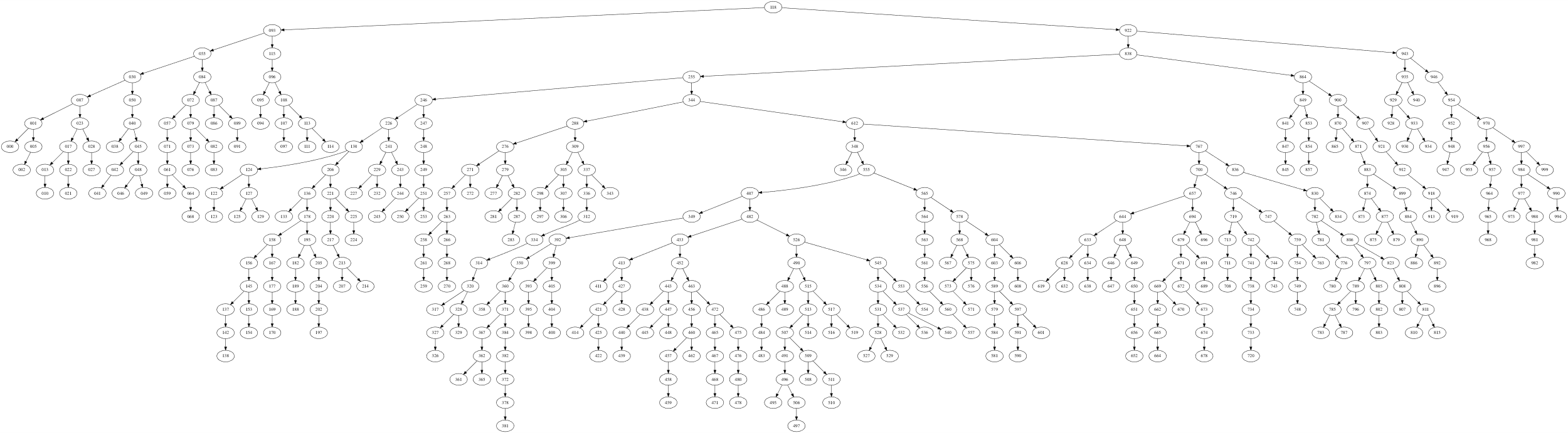


Рисунок 4 – Дерево из 500 элементов

* 1. **Вывод**

В ходе выполнения лабораторной работы была написана программа, реализующая построение бинарных деревьев поиска и линейных списков на основании данных, расположенных в файле. Было выполнено исследование структур данных бинарных деревьев поиска и линейных списков и вычислено среднее время выполнения операций добавления, удаления и поиска информации.

**ПРИЛОЖЕНИЕ А**

Листинг 1 - Текст программы с реализацией классов для работы с деревьями и со списками.

#include <iostream>

#include <windows.h>

#include <fstream>

#include <random>

#include <ctime>

#include <vector>

#include <chrono>

#include <iomanip>

#include <string>

#include "Data.hpp"

using namespace std;

class ListNode {

private:

Data data;

ListNode\* next;

public:

ListNode(Data value) : data(value), next(nullptr) {}

ListNode\* getNext() { return next; }

void setNext(ListNode\* node) { next = node; }

Data getData() { return data; }

};

class List {

private:

ListNode\* begin;

ListNode\* end; // Новый указатель на конец списка

public:

List() : begin(nullptr), end(nullptr) {}

void add(Data value) {

ListNode\* newNode = new ListNode(value);

if (begin == nullptr) { // Если список пуст

begin = newNode;

end = newNode; // Устанавливаем конец на первый элемент

}

else {

end->setNext(newNode); // Присоединяем новый элемент в конец

end = newNode; // Обновляем конец списка

}

}

void displayAsGraph() {

ofstream file("list.dot", ios::out);

if (!file) {

cerr << "Error opening file." << endl;

return;

}

file << "digraph List {" << endl;

ListNode\* current = begin;

while (current != nullptr) {

file << current->getData().subscriber << ";" << endl;

if (current->getNext() != nullptr) {

file << current->getData().subscriber << " -> " << current->getNext()->getData().subscriber << ";" << endl;

}

current = current->getNext();

}

file << "}" << endl;

file.close();

cout << "List saved in file: list.dot" << endl;

}

ListNode\* searchElement(string subscriber) {

ListNode\* element = begin;

for (; element != nullptr; element = element->getNext()) {

if (element->getData().subscriber == subscriber) {

return element;

}

}

return nullptr;

}

void deleteElement(string subscriber) {

if (!begin) {

cout << "The list is empty." << endl;

return;

}

if (begin->getData().subscriber == subscriber) {

ListNode\* temp = begin;

begin = begin->getNext();

delete temp;

if (begin == nullptr) { // Если удален последний элемент

end = nullptr; // Обнуляем указатель на конец

}

return;

}

ListNode\* current = begin;

while (current->getNext() != nullptr) {

if (current->getNext()->getData().subscriber == subscriber) {

ListNode\* temp = current->getNext();

current->setNext(current->getNext()->getNext());

delete temp;

if (current->getNext() == nullptr) { // Если удален последний элемент

end = current; // Обновляем указатель на конец

}

return;

}

current = current->getNext();

}

cout << "Element not found." << endl;

}

void printTimeTable() {

double addTime = 0.0, deleteTime = 0.0, searchTime = 0.0;

for (int i = 0; i < 5; i++) {

auto start = chrono::high\_resolution\_clock::now();

add(generateRandomData(i));

auto end = chrono::high\_resolution\_clock::now();

double oneElementAddTime = chrono::duration<double, milli>(end - start).count();

cout << "Addition #" << i + 1 << ": " << oneElementAddTime << " ms" << endl;

addTime += oneElementAddTime;

start = chrono::high\_resolution\_clock::now();

searchElement(generateRandomString());

end = chrono::high\_resolution\_clock::now();

double oneElementSearchTime = chrono::duration<double, milli>(end - start).count();

cout << "Search #" << i + 1 << ": " << oneElementSearchTime << " ms" << endl;

searchTime += oneElementSearchTime;

start = chrono::high\_resolution\_clock::now();

deleteElement(generateRandomString());

end = chrono::high\_resolution\_clock::now();

double oneElementDeleteTime = chrono::duration<double, milli>(end - start).count();

cout << "Deletion #" << i + 1 << ": " << oneElementDeleteTime << " ms" << endl;

deleteTime += oneElementDeleteTime;

}

cout << "Average operation times (in ms):" << endl;

cout << "===================================" << endl;

cout << "| Operation | Avg. time |" << endl;

cout << "===================================" << endl;

cout << "| Add | " << fixed << setprecision(3) << setw(10) << addTime / 5 << " |" << endl;

cout << "| Search | " << fixed << setprecision(3) << setw(10) << searchTime / 5 << " |" << endl;

cout << "| Delete | " << fixed << setprecision(3) << setw(10) << deleteTime / 5 << " |" << endl;

cout << "===================================" << endl;

}

};

class TreeNode {

private:

Data data;

TreeNode\* left;

TreeNode\* right;

public:

TreeNode(Data value) : data(value), left(nullptr), right(nullptr) {}

TreeNode\* getLeft() { return left; }

TreeNode\* getRight() { return right; }

void setLeft(TreeNode\* node) { left = node; }

void setRight(TreeNode\* node) { right = node; }

Data getData() { return data; }

};

class Tree {

private:

TreeNode\* root;

public:

Tree() : root(nullptr) {}

void add(Data value) {

if (root == nullptr) {

root = new TreeNode(value);

}

else {

addRecursively(root, value);

}

}

void addRecursively(TreeNode\* node, Data value) {

if (value.subscriber < node->getData().subscriber) {

if (node->getLeft() == nullptr) {

node->setLeft(new TreeNode(value));

}

else {

addRecursively(node->getLeft(), value);

}

}

else {

if (node->getRight() == nullptr) {

node->setRight(new TreeNode(value));

}

else {

addRecursively(node->getRight(), value);

}

}

}

void displayAsGraph() {

ofstream file("tree.dot", ios::out);

if (!file) {

cerr << "Error opening file." << endl;

return;

}

file << "digraph Tree {" << endl;

displayNodeAsGraph(file, root);

file << "}" << endl;

file.close();

cout << "Tree saved in file: tree.dot" << endl;

}

void displayNodeAsGraph(ofstream& file, TreeNode\* node) {

if (node == nullptr) {

return;

}

// Записываем текущий узел в файл

file << node->getData().subscriber << ";" << endl;

// Если у узла есть левый дочерний элемент, записываем связь

if (node->getLeft() != nullptr) {

file << node->getData().subscriber << " -> " << node->getLeft()->getData().subscriber << ";" << endl;

displayNodeAsGraph(file, node->getLeft());

}

// Если у узла есть правый дочерний элемент, записываем связь

if (node->getRight() != nullptr) {

file << node->getData().subscriber << " -> " << node->getRight()->getData().subscriber << ";" << endl;

displayNodeAsGraph(file, node->getRight());

}

}

// Поиск элемента в дереве

TreeNode\* searchElement(string subscriber) {

return searchRecursive(root, subscriber);

}

TreeNode\* searchRecursive(TreeNode\* node, string subscriber) {

if (node == nullptr || node->getData().subscriber == subscriber) {

return node;

}

if (subscriber < node->getData().subscriber) {

return searchRecursive(node->getLeft(), subscriber);

}

else {

return searchRecursive(node->getRight(), subscriber);

}

}

// Удаление элемента из дерева

void deleteElement(string subscriber) {

root = deleteRecursive(root, subscriber);

}

TreeNode\* deleteRecursive(TreeNode\* node, string subscriber) {

if (node == nullptr) {

return node;

}

if (subscriber < node->getData().subscriber) {

node->setLeft(deleteRecursive(node->getLeft(), subscriber));

}

else if (subscriber > node->getData().subscriber) {

node->setRight(deleteRecursive(node->getRight(), subscriber));

}

else {

if (node->getLeft() == nullptr) {

TreeNode\* temp = node->getRight();

delete node;

return temp;

}

else if (node->getRight() == nullptr) {

TreeNode\* temp = node->getLeft();

delete node;

return temp;

}

TreeNode\* temp = findMin(node->getRight());

node->getData() = temp->getData();

node->setRight(deleteRecursive(node->getRight(), temp->getData().subscriber));

}

return node;

}

TreeNode\* findMin(TreeNode\* node) {

TreeNode\* current = node;

while (current && current->getLeft() != nullptr) {

current = current->getLeft();

}

return current;

}

void printTimeTable() {

double addTime = 0.0, deleteTime = 0.0, searchTime = 0.0;

for (int i = 0; i < 5; i++) {

auto start = chrono::high\_resolution\_clock::now();

add(generateRandomData(i));

auto end = chrono::high\_resolution\_clock::now();

double oneElementAddTime = chrono::duration<double, milli>(end - start).count();

cout << "Addition #" << i + 1 << ": " << oneElementAddTime << " ms" << endl;

addTime += oneElementAddTime;

start = chrono::high\_resolution\_clock::now();

searchElement(generateRandomString());

end = chrono::high\_resolution\_clock::now();

double oneElementSearchTime = chrono::duration<double, milli>(end - start).count();

cout << "Search #" << i + 1 << ": " << oneElementSearchTime << " ms" << endl;

searchTime += oneElementSearchTime;

start = chrono::high\_resolution\_clock::now();

deleteElement(generateRandomString());

end = chrono::high\_resolution\_clock::now();

double oneElementDeleteTime = chrono::duration<double, milli>(end - start).count();

cout << "Deletion #" << i + 1 << ": " << oneElementDeleteTime << " ms" << endl;

deleteTime += oneElementDeleteTime;

}

cout << "Average operation times (in ms):" << endl;

cout << "===================================" << endl;

cout << "| Operation | Avg. time |" << endl;

cout << "===================================" << endl;

cout << "| Add | " << fixed << setprecision(3) << setw(10) << addTime / 5 << " |" << endl;

cout << "| Search | " << fixed << setprecision(3) << setw(10) << searchTime / 5 << " |" << endl;

cout << "| Delete | " << fixed << setprecision(3) << setw(10) << deleteTime / 5 << " |" << endl;

cout << "===================================" << endl;

}

};

void addMultipleElements(int elementsAmount, Tree& tree) {

for (int i = 0; i < elementsAmount; i++) {

Data data = generateRandomData(i);

tree.add(data);

}

}

void addMultipleElements(int elementsAmount, List& list) {

for (int i = 0; i < elementsAmount; i++) {

Data data = generateRandomData(i);

list.add(data);

}

}

int main() {

List list;

Tree tree;

int elementsAmount;

cout << "How many elements to add?" << endl << "Amount: ";

cin >> elementsAmount;

cout << "List:" << endl;

addMultipleElements(elementsAmount, list);

list.printTimeTable();

list.displayAsGraph();

cout << "Tree:" << endl;

addMultipleElements(elementsAmount, tree);

tree.printTimeTable();

tree.displayAsGraph();

return 0;

}

#ifndef DATA\_HPP

#define DATA\_HPP

#include <iostream>

#include <string>

#include <random>

#include <vector>

struct Data {

int ip;

std::string subscriber;

std::string phoneNumber;

std::string name;

int amount;

friend std::ostream& operator<<(std::ostream& os, const Data& data);

friend std::istream& operator>>(std::istream& is, Data& data);

};

std::ostream& operator<<(std::ostream& os, const Data& data) {

os << data.ip << " " << data.subscriber << " " << data.phoneNumber << " " << data.name << " " << data.amount;

return os;

}

std::istream& operator>>(std::istream& is, Data& data) {

is >> data.ip >> data.subscriber >> data.phoneNumber >> data.name >> data.amount;

return is;

}

std::string generateRandomPhoneNumber() {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(0, 9);

std::string phoneNumber = "";

for (int i = 0; i < 10; i++) {

if (i == 0) {

phoneNumber += std::to\_string(dis(gen) + 1);

}

else {

phoneNumber += std::to\_string(dis(gen));

}

}

return "+7" + phoneNumber;

}

std::vector<std::string> firstNames = {

"Liam", "Emma", "Noah", "Olivia", "William", "Ava", "James", "Isabella",

"Oliver", "Sophia", "Benjamin", "Mia", "Elijah", "Charlotte", "Lucas", "Harper"

};

std::vector<std::string> lastNames = {

"Smith", "Johnson", "Williams", "Brown", "Davis", "Miller", "Wilson", "Moore",

"Taylor", "Anderson", "Thomas", "Jackson", "White", "Harris", "Martin", "Thompson"

};

std::string generateRandomName() {

static std::random\_device rd;

static std::mt19937 gen(rd());

static std::uniform\_int\_distribution<> firstNameDist(0, firstNames.size() - 1);

static std::uniform\_int\_distribution<> lastNameDist(0, lastNames.size() - 1);

std::string firstName = firstNames[firstNameDist(gen)];

std::string lastName = lastNames[lastNameDist(gen)];

return firstName + " " + lastName;

}

int generateRandomNumber() {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(100000, 999999);

return dis(gen);

}

std::string generateRandomString() {

static std::random\_device rd;

static std::mt19937 gen(rd());

static std::uniform\_int\_distribution<> charDist(48, 57);

std::string randomString;

for (int i = 0; i < 3; i++) {

randomString += static\_cast<char>(charDist(gen));

}

return randomString;

}

Data generateRandomData(int i) {

Data data;

data.ip = i + 1;

data.subscriber = generateRandomString();

data.phoneNumber = generateRandomPhoneNumber();

data.name = generateRandomName();

data.amount = generateRandomNumber();

return data;

}

#endif // DATA\_HPP